A CGAL-based Univariate Algebraic Kernel and Application to Arrangements

Sylvain Lazard, Luis Mariano Peñaranda, Elias P. P. Tsigaridas

A CGAL-based Univariate Algebraic Kernel and Application to Arrangements

Sylvain Lazard[1], Luis Peñaranda[1], and Elias Tsigaridas[2⋆]

[1] LORIA (INRIA, CNRS, Nancy Université) and INRIA Nancy - Grand Est, France.
FirstName.LastName@loria.fr
[2] INRIA Sophia-Antipolis - Méditerranée, France.
FirstName.LastName@sophia.inria.fr

Abstract. Solving univariate polynomials and operations with real algebraic numbers are critical in geometric computing with curved objects. Moreover, the real roots need to be computed in a certified way in order to avoid possible inconsistency in geometric algorithms. We present a CGAL-based univariate algebraic kernel, which follows the CGAL specifications for univariate kernels. It provides certified real-root isolation of univariate polynomials with integer coefficients (based on the library RS) and standard functionalities such as basic arithmetic operations, gcd and square-free factorization, as well as comparison and sign evaluations of real algebraic numbers. We compare our implementation with the univariate algebraic kernel developed at MPII, that also follows CGAL specifications, on various data sets, using polynomials of degree up to 2 000 and maximum coefficient bit-size up to 25 000. Finally we apply our kernel to the computation of arrangements of univariate polynomial functions, and we present a bit complexity analysis of the algorithm for computing the arrangement of planar curves defined by univariate polynomials.

1 Introduction

Implementing geometric algorithms robustly is known to be a difficult task for two main reasons. First, all degenerate situations have to be handled and second, algorithms often assume a real-RAM model which is not realistic in practice. In recent years, the paradigm of exact geometric computing arose as a standard for robust implementations. In this paradigm, geometric decisions, such as "is a point inside, outside or on a circle?", are made exactly, usually using either exact arithmetic combined, for efficiency, with interval arithmetic (on doubles) or using interval arithmetic (on arbitrary-fixed-precision floating-point numbers) combined with separation bounds ; on the other hand, geometric constructions, such as the coordinates of a point of intersection, may be approximated.

We address here one recurrent difficulty arising when implementing algorithms dealing with curved objects. Such algorithms usually require evaluating,

⋆ Most part of this work was done while the author was at LORIA - INRIA Nancy Grand Est.

manipulating and solving systems of polynomials equations and comparing their roots. One of the most critical parts of dealing with polynomials or polynomial systems is the isolation of the real roots and their comparison.

We restrict here our attention to the case of univariate polynomials and address this problem in the context of CGAL, a C++ Computational Geometry Algorithms Library, which is an open source project and became a standard for the implementation of geometric algorithms [5].

Our effort is towards introducing to CGAL a kernel capable of dealing with non linear objects. The combination of geometry and algebra for effective manipulation of non linear objects is a long standing challenge. Previous work includes, but it is not limited to, MAPC [22] a library for handling curves in the plane and its more recent successor ESOLID [21]. For other efforts more closely related to CGAL we should also mention [26] where the problem of computing the arrangement of conics arcs in the plane is considered, and the algebraic operations needed, operations with algebraic numbers of degree up to 4, were based on CORE [20]. The notion of the algebraic kernel for CGAL was proposed in [15] where the underlying algebraic operations were based on SYNAPS library [23]. A complete design and a software library, namely EXACUS, for supporting algebraic operations in non linear computational geometry proposed in [7]. Recent advances in non linear computational geometry are presented in [9].

CGAL is designed in a modular fashion. Algorithms are typically parametrized by a *traits* class which encapsulates the geometric objects, predicates and constructions used by the algorithm. Typically, this allows implementing algorithms independently of the type of input objects. For instance, a sweep-line algorithm for computing arrangements can be implemented generically for segments or curves. Similarly, the model of computation, such as exact arbitrary-length integer arithmetic or approximate fixed-precision floating-point arithmetic are encapsulated in the concept of *kernel*. An implementation is thus typically separated in three layers, the geometric algorithm which relies on a traits class, which itself relies on a kernel for elementary operations. A choice of traits class and kernel gives freedom to the users and allows comparison.

Our Contribution. The contributions of this paper are the following: We propose a CGAL compliant, based on the specifications in [8], open source algebraic kernel based on the library RS [25] that provides real root isolation of univariate rational polynomials and basic operations, i.e. comparisons and sign evaluations, of real algebraic numbers. Moreover, our kernel provides various operations on polynomials, such as gcd, which are central for manipulating algebraic numbers. Our code will be submitted to the CGAL's editorial board in the near future.

Second, we illustrate the efficiency of our code on various data sets, concerning real root isolation of univariate polynomials of degree up to 2 000 and bit-size up to 25 000 and comparison of real algebraic numbers in several configurations. We test our code against the kernel developed by Hemmer and Limbach [19].

Finally, we sketch our traits class for computing arrangements of polynomial functions and we present an output sensitive bit-complexity bound, for computing the arrangement of planar curves defined by univariate integer polynomials.

We establish a bound $\widetilde{\mathcal{O}}_B\left((n+k)d^2(d\tau+ds+t^2+s^2)\right)$, where n is the number of curves, d and τ bound the degree and the maximum coefficient bit-size of the polynomials respectively, and k is the number of intersections. The $\widetilde{\mathcal{O}}_B(\cdot)$ notation ignores the logarithmic factors. To the best of our knowledge this is the first time the bit complexity of the algorithm is considered, let alone an output sensitive version of it.

The rest of the paper is structured as follows. In the next section we describe our univariate algebraic kernel. In Section 3 we present various experiments concerning real root isolation and comparison of real algebraic numbers. Finally in Section 4 we sketch our traits class for arrangements and we present the bit complexity analysis of the algorithm for computing the arrangement of curves defined by univariate polynomials.

2 Univariate algebraic kernel

We describe here our implementation of our univariate algebraic kernel. The two main requirements of the CGAL specifications, which we describe here, are the isolation of real roots and their comparison. We also describe our implementation of two important specific operations, greatest common divisor (gcd) computation and refinement of isolating intervals, that are needed, in particular, for comparing algebraic numbers.

Preliminaries. The kernel handles univariate polynomials and algebraic numbers. The polynomials have integer coefficients and are represented by arrays of GMP arbitrary-length integers [2]. We implemented in the kernel the basic functions on polynomials, including basic arithmetic, evaluation, and input/output. An algebraic number that is a root of a polynomial F is represented by F and an isolating interval, that is an interval containing this root but no other root of F. We implemented intervals using the MPFI library [4], which represents intervals with two MPFR arbitrary-fixed-precision floating-point numbers [3]; note that MPFR is developed on top of the GMP library for multi-precision arithmetic [2].

Root isolation. For isolating the real roots of univariate polynomials with integer coefficients, we developed an interface with the library RS [25]. This library is written in C and is based on Descartes' rule for isolating the real roots of univariate polynomials with integer coefficients.

We briefly detail here the general design of the RS library; see [24] for details. RS is based on an algorithm known as *interval Descartes* [10]; namely, the coefficients of the polynomials obtained by changes of variable, sending intervals $[a,b]$ onto $[0,+\infty]$, are only approximated using interval arithmetic when this is sufficient for determining their signs. Note that the order in which these transformations are performed in RS is important for memory consumption. The intervals and operations on them are handled by the MPFI library. Another characteristic of RS is its memory management: it implements a *mark-and-sweep* garbage collector, which is well suited to RS needs.

Algebraic number comparison. As mentioned above, one of the main requirements of the CGAL algebraic kernel specifications is to compare two algebraic numbers r_1 and r_2. If we are lucky, their isolating intervals do not overlap and the comparison is straightforward. This is, of course, not always the case. If we knew that they were not equal, we could refine both isolating intervals until they are disjoint; see below for details on how we perform the refinements. Hence, the problem reduces to determining whether the algebraic numbers are equal or not.

To do so, we compute the square free factorization of the gcd of the polynomials P_1 and P_2 associated to the algebraic numbers; see below for details on this operation. The roots of this gcd are the common roots of both polynomials. We calculate the intersection, I, of the isolating intervals of r_1 and r_2. The gcd has a root in this interval if and only if $r_1 = r_2$.

To determine whether the gcd has a root in interval I, it suffices to check the sign of the gcd on the endpoints of I: if they are different or one of them is zero, the gcd has a root in I and $r_1 = r_2$; otherwise, $r_1 \neq r_2$ and we can refine both intervals until they are disjoint.

Gcd computations. Computing greatest common divisors between two polynomials is not a difficult task, however, it is not trivial to do so efficiently. Indeed, a naive implementation of the Euclidean algorithm works fine for small polynomials but the intermediate coefficients suffer an exponential grow in size, which is not manageable for medium to large size polynomials.

We thus implemented a *modular* gcd function, which calculates the gcd of polynomials modulo some prime numbers and reconstructs later the result with the help of the *Chinese remainder theorem*. Details on these algorithms can be found in [18]. Note that modular gcd is always more efficient than regular gcd and it is much more efficient when the two polynomials have no common roots.

Refining isolating intervals. As we mentioned before, refining the interval representing an algebraic number is critical for comparing such numbers. We have implemented two approaches for refinement.

Both approaches require that the polynomial associated to the algebraic number is square free. The first step thus consists of computing the square-free part of the polynomial. This is easily done by computing the gcd of the polynomial and its derivative.

Our first approach is a simple bisection algorithm. It consists in calculating the sign of the polynomial associated to the algebraic number at the endpoints and midpoint of the interval. Depending on those three signs we can take as isolating interval the left of right half of the previous one.[3]

The second approach we implemented is the *quadratic interval refinement* [6]. Roughly speaking, this method splits the interval in many parts and, based on a linear interpolation, guesses in which one the root lies. If the guess is correct, the algorithm will divide, in the next refinement step the (chosen) interval in more

[3] Note that since the polynomial is square free the signs at the two endpoints of any isolating interval always differ. We thus do not need to compute the sign at both endpoints.

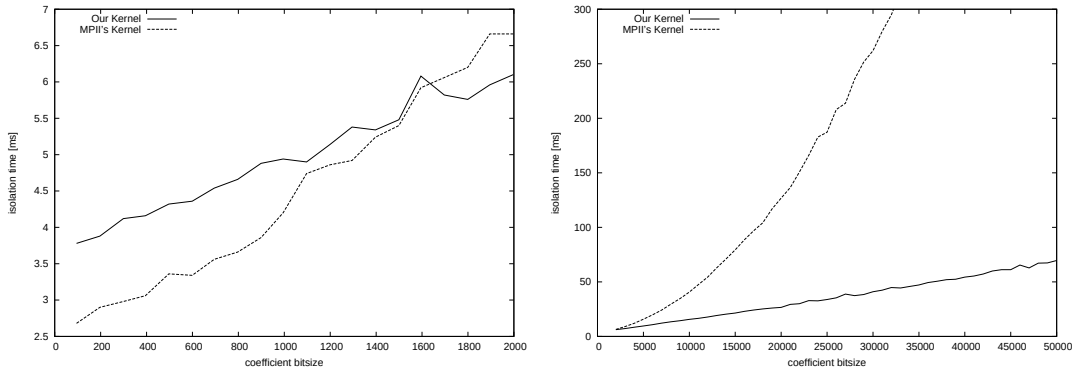


Fig. 1. Running time for isolating all the real roots of degree 12 polynomials with 12 real roots in terms of the maximum bit-size of their coefficients.

parts and, if not, in less. To be efficient, this approach requires the development of functions to handle dyadic numbers efficiently.[4] Note that these functions are also useful in the bisection method when increasing the precision (because working directly with MPFR is rather tricky). Unfortunately, even with a careful implementation this approach turns out to be, on average, only just a bit faster than the bisection approach.

Currently, refinement function based on both approaches are present in our kernel and the user can choose the one best suited to her/his needs.

3 Kernel benchmarks

In this section, we analyze the running time of the two main functions of our algebraic kernel, that is (i) isolating the roots of a polynomial and (ii) comparing two algebraic numbers that is, comparing the roots of two polynomials. We also compare the performance of our kernel with the one developed by Hemmer and Limbach [19] (which we refer to as the MPII's kernel).

All tests were ran on a single-core 3.2 GHz Intel Pentium 4 with 2 Gb of RAM, using 64-bit Linux.

Root isolation. We consider two suites of experiments in which we either fix the degree of the polynomials and vary the bit-size of the coefficients or the converse; see Figures 1 and 2. In each experiment, we report the running time for isolating all the roots per polynomial, averaged over different trials, for both our kernel and MPII's kernel.

[4] This was coded keeping the GMP flavor of all libraries we work with and operating with GMP's limbs (`mpn_t`, the lowest abstraction level this library offers) when needed.

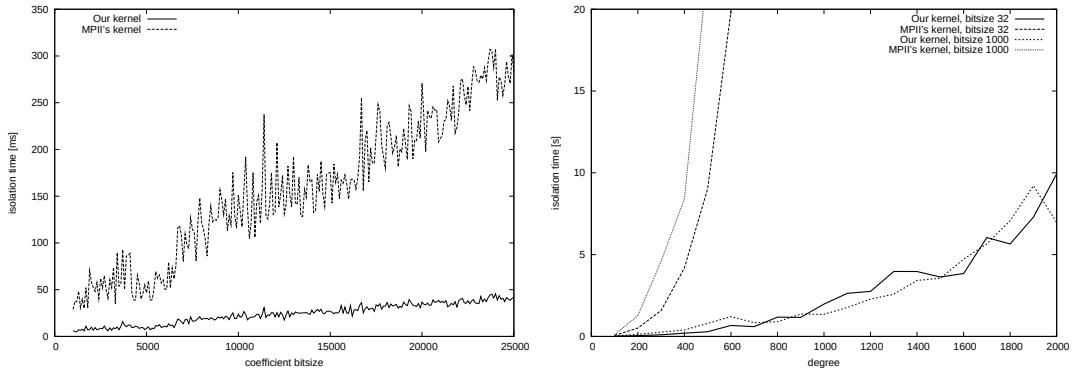


Fig. 2. Running time for isolating all the real roots of (a) degree 100 polynomials in terms of the maximum bit-size of their coefficients and (b) random polynomials with coefficients of maximum bit-size 20 000 and depending on the degree.

Varying bit-size. We study here polynomials with rather low degree (12) but with no complex root and polynomials with reasonably large degree (100) with random coefficients (and thus with few roots).

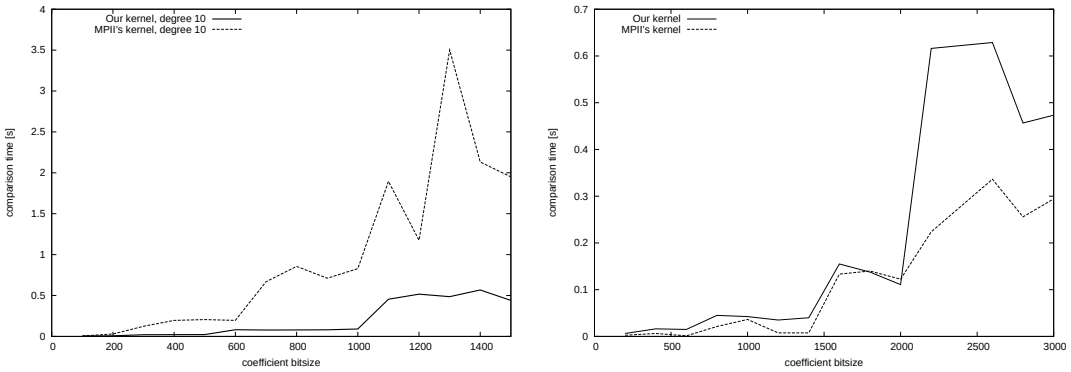
The first test sets comes from [19]. See Figure 1. It consists of polynomials of degree 12, each one being the product of six degree-two polynomials that have at least a root in the interval $[0, 1]$; every polynomial thus has 12 real roots. We vary the maximum bit-size of all the coefficients of the input polynomial from 100 to 50 000 and average each test over 50 trials.

Secondly, we consider random polynomials with constant degree 100 and coefficients with varying bit-size. See Figure 2(a). Note that such random polynomials have few roots: the expected number of real roots of a polynomial of degree d with coefficients independently chosen from the standard normal distribution is $\frac{2}{\pi}\log(d) + C + \frac{2}{\pi d} + O(1/d^2)$ where $C \approx 0.625735$ [13]; this gives, for degree 100 an average of about 3.6 roots (note that this bound matches extremely well experimental observations). We vary the maximum bit-size of all the coefficients from 2 000 to 25 000 and average each test over 100 trials.

Varying degree. We consider two sets of experiments in which we study random polynomials and Mignotte polynomials (which have two very close roots).

We first consider polynomials with random coefficients of maximum bit-size 20 000. We then vary the degree of the polynomials from 100 to 2 000. Note that the above formula gives an expected number of roots varying from 3.6 to 5.5. The results, shown in Figure 2(b) are averaged over 100 trials.

Finally, we test Mignotte polynomials, that is nearly degenerate polynomials of the form $x^d - 2(kx-1)^2$. The difficulty with solving these polynomials lies in the fact that two of their roots are very close to each other (the isolating intervals for these two roots are thus very small). For these tests, we used Mignotte polynomials with coefficients of bit-size 50, with varying degree d from 5 to 50. We averaged the running times over 5 trials for each degree. We observed



(a) (b)

Fig. 3. (a) Running time for comparing two close roots of two almost identical polynomials of degree 10 with no common roots. (b) Running time for comparing two close or equal roots of two polynomials defined as the product of two degree-10 factors, one in common and the other being almost identical in the two polynomials.

essentially no difference between the two tested kernels; they take roughly 0.2 and 5.5 seconds for Mignotte polynomials of degree 20 and 50, respectively. Because of lack of space, we do not present the running-time graphs.

Conclusion. Figure 1(a) shows that our kernel's performance is worse than MPII's one for small degree polynomials. This difference comes from the fact that RS, the most consuming part of our process, is conceived for handling large polynomials. This fact is confirmed by Figures 1(b), 2(a) and 2(b), which show that for polynomials with larger coefficients or higher degree, our kernel runs faster.

We also observe that the running time of our kernel is very stable for isolating roots and does not depend too much on the bit-size or on the degree of the input polynomials.

Comparison of algebraic numbers. We consider three suites of experiments for comparing algebraic numbers; see Figures 3. Recall that an algebraic number ρ is here represented by a polynomial F that vanishes at ρ and an isolating interval containing ρ but no other root of F. Recall also that the comparison of two algebraic numbers is done by (i) testing whether the interval overlap are disjoint; if so, report the ordering, otherwise (ii) compute the gcd of the two polynomials and test whether the gcd vanishes in the intersection of the two intervals; if so, report the equality of the numbers, otherwise (iii) refine the intervals until they are disjoint.

First, we analyse the cost of trivial comparisons that is, when the two intervals representing the numbers are disjoint. For that we compare the roots of two random polynomials. We observe that, as expected, the comparison time is negligible and independent of both the degree of the polynomials and the bit-size of their coefficients.

Second, we analyze the cost of comparing roots that are very close to each others but whose associate polynomials have no common root. This case is expensive because we need to refine the intervals until they do not overlap; this is, however, not the worse situation because the gcd of the two polynomials is 1 which is tested efficiently with a modular gcd. We perform these experiments as follows. We generate pairs of polynomials, one with random coefficients and the other by only adding 1 to one of the coefficients of the first polynomial. Such polynomials are such that the i-th roots of both polynomials are very close to each other. We generate such pairs of polynomials with constant degree (equal to 10) and vary the maximum bit-size of the coefficients. As the bit-size increases, the pairs of roots that are close become even closer and thus the comparison time increases. The results of these experiments are presented in Figure 3(a), which reports the average running time for comparing two close roots.[5]

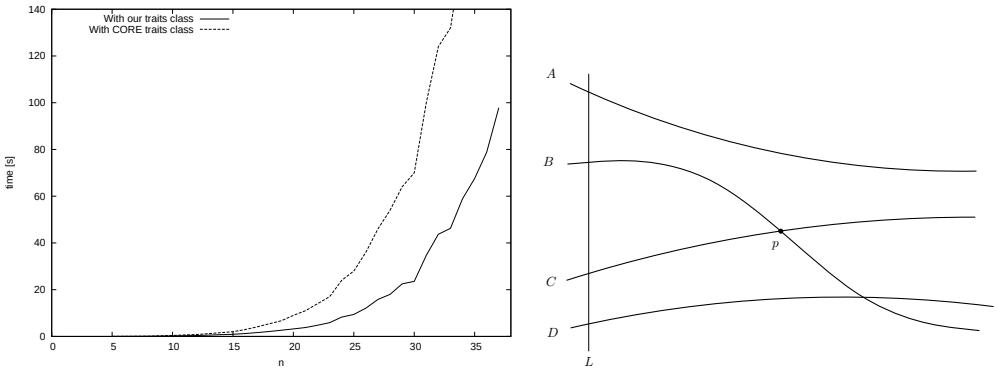
Third, we consider the, a priori, most expensive scenario in which we compare roots that are either equal or very close to each others and such that their associate polynomials have some roots in common. In this case, we cumulate the cost of computing a non-trivial gcd of the two polynomials with the cost of refining intervals when comparing two non-equal roots. In practice, we generate pairs of degree-20 polynomials each defined as the product of two degree-10 terms; one of these factors is random and common to the two polynomials; the other factor is random in one of the polynomials and slightly modified in the other polynomial where, slightly modified means, as above, that we add 1 to one of the coefficients. We then vary the maximum bit-size of the coefficients and average each test over four trials.

Conclusion. We observe that for comparing roots of small degree polynomials with no common roots, our kernel performs better than the MPII's kernel. However, the MPII's kernel performs slightly better for comparing roots of small degree polynomials with roots in common. Surprisingly, the cost of comparing roots of polynomials with a common factor is not more expensive than when the polynomials have no root in common. This might be due to the fact that we average the cost of comparison over all non-trivial comparisons (that is over the number of real roots of each polynomial) and it is possible that the comparison of two equal roots (that is the cost of the gcd) is almost negligible compared to the comparison of two close roots (that is the cost of the gcd plus the interval refinements). Overall, it appears that comparing algebraic number that are very close is fairly time consuming.

4 Arrangements

As an example of possible benefit of having efficient algebraic kernels in CGAL, we used our implementation to construct arrangements of polynomial functions. Wein and Fogel provided a CGAL package for calculating arrangements [27].

[5] According to our first set of experiments, we can neglect the time for comparing two roots that are not close.



(a) (b)

Fig. 4. (a) Running time for computing arrangements of n graphs of polynomials of degree $n-1$ with (n) coefficients of bit-size n. (b) Arrangement of planar curves. The sweep algorithm starts from line L.

This package calculates the arrangements of general curves [17]. It is the user who must implement the data structures to store the curves and the primitive operations; requiring for example comparing positions of points, comparing the vertical order of curves at infinity and intersecting and splitting curves. All these functions must be grouped in a *traits class*, which is a transparent and convenient way to work with a package in CGAL. We implemented a traits class which uses the functions of our algebraic kernel and compared its performance with another traits classes which comes with CGAL's arrangement package and uses the CORE library [1].

To test the arrangement calculation, we generated n polynomials of degree $n-1$ with (n) coefficients of bit-size n. The running time for the construction of this type of arrangements is shown in Figure 4(a). We observe that we gain a factor of roughly two when using our kernel.[6]

Complexity Analysis. In this section we present an output sensitive analysis of the bit complexity of the problem of computing the arrangement of univariate polynomial functions. The same analysis, and thus the same complexity bound, applies also in the case of rational univariate functions. In what follows arithmetic, respectively bit complexity will be denoted by $\mathcal{O}$, respectively $\mathcal{O}_B$. The $\widetilde{\mathcal{O}}$ and $\widetilde{\mathcal{O}}_B$ notation means that we are ignoring (poly-)logarithmic factors. For an integer polynomial $f \in \mathbb{Z}[x]$, $\deg(f)$ denotes its degree and $\mathcal{L}(f)$ the maximum bit-size of its coefficients (including a bit for the sign). We will need the following results.

[6] Note to the reviewers: the experiments presented here are not up to date. We performed these experiments in last Sept. with a preliminary version of our kernel. We have since substantially improved it but the CGAL package for calculating arrangements has also substantially changed implying some important changes to our traits class. We are in the process of updating our traits class with which we will be able to update our experiments and tests degenerate and near-degenerate arrangements.

Proposition 1. *[14, 16] We can isolate, using either Sturm, Descartes or Bernstein algorithm, the real roots of $f \in \mathbb{Z}[x]$, where $\deg(f) = d$ and $\mathcal{L}(f) = \tau$, and compute their multiplicities in $\widetilde{\mathcal{O}}_B(d^6 + d^4\tau^2)$. The bit-size of the endpoints of the isolating intervals is $\mathcal{O}(d^2 + d\tau)$.*

Proposition 2. *[12] We can compare, using either Sturm-Habicht sequences or refinements and GCD computation, two real algebraic numbers defined by polynomials with degrees bounded by d and bit-size bounded by τ in $\widetilde{\mathcal{O}}_B(d^3\tau)$.*

Remark 1. We can modify the proof of Prop. 1 so that to express the complexity with respect to the logarithm of the actual separation bound, i.e. the bit-size of the minimum distance between two (possible complex) roots of the polynomial and not the theoretical one. If we denote this by s, then the previous bound becomes $\widetilde{\mathcal{O}}_B(d^3(\tau^2+s^2))$, which can be considered as output sensitive complexity. Under this notion the bit-size of the endpoints of the isolating intervals becomes $\widetilde{\mathcal{O}}(s)$. Moreover, comparison of two real algebraic numbers can be performed in $\widetilde{\mathcal{O}}_B(d^2(\tau + s))$.

In practice the worst case complexity bounds of Prop. 1 and Prop. 2 as well as the output sensitive bound of Rem. 1 are a big overestimation.

Recall [11] that the arithmetic complexity of computing the arrangement is $\mathcal{O}((n + k)\log n)$, where n is the number of curves, and k is the number of intersection points. The reader may refer to [17] for a complete description of the algorithm. We split the cost of the algorithm to two main parts. The first one is to construct the intersection points of the curves and the second part represents the cost of comparing them. We consider each part independently.

In order to compute the intersections of two curves $y = f_1(x)$ and $y = f_2(x)$, represented by polynomials $f_1, f_2 \in \mathbb{Z}[x]$, of degree bounded by d and maximum coefficient bit-size bounded by τ, it suffices to compute the real roots of the polynomial $f(x) = f_1(x) - f_2(x)$, which is of degree $\mathcal{O}(d)$ and bit-size $\mathcal{O}(\tau)$. This can be done in $\widetilde{\mathcal{O}}_B(d^3(\tau^2 + s^2))$ (Rem. 1), where s is the bit-size of the (actual) separation bound (i.e. the smallest distance between two roots of f).

Let $\#(S)$ be the number of times that we need to perform a solve operation. Evidently, $\#(S) \leq n(n-1)/2$, since in the worst case we have to consider the intersections of all possible pairs of the n curves. However, it holds that $\#(S) \leq n + 2k$, which is an output sensitive result.

To see this, we will use the help of Fig. 4(b). To begin the algorithm we need to compute a vertical line [7], Line L in Fig. 4(b), to the left of all the intersection points and to compute the intersections of this line with all the curves, so that to provide the algorithm with initial points. Thus, initially we need to perform n solve operations. In the sequel, for each intersection point we encounter, say the point p which is an intersection point of curves B and C in Fig. 4(b), we proceed as follows. We can consider this point as the left endpoint of the curve segments of B and C starting from it. For B, resp. C, we have to compute its

[7] The cost of computing the line is $\widetilde{\mathcal{O}}_B(n\,d\,\tau)$ and is dominated by the other steps of the algorithm.

intersections with the curves that are above and below of it, just after p. These are C and D, resp. A and B. Since we have already computed the intersections with C, resp. B, we just need to compute the intersections with D, resp. A. Thus for each intersection point we need to perform at most 2 solve operations, and in total we perform $n + 2k$, or $\mathcal{O}(n + k)$.

The second part of the cost of the algorithm is the cost of the $\mathcal{O}((n+k)\log n)$ comparisons. Each comparison corresponds to a comparison of two real algebraic numbers, defined by polynomials of degree bounded by d and bit-size bounded by τ. This can be done in $\widetilde{\mathcal{O}}_B(d^2(\tau + s))$ (Rem. 1).

We can state the following theorem

Theorem 1. *We can compute the arrangement of n curves, defined by univariate polynomials of degree bounded by d and maximum coefficient bit-size bounded by τ in $\widetilde{\mathcal{O}}_B\left((n+k)d^2(d\tau + ds + t^2 + s^2)\right)$, where k is the number of intersections points between the polynomials.*

If $N = \max\{n, k, d, \tau, s\}$ then the previously complexity becomes $\widetilde{\mathcal{O}}_B(N^6)$, which is not as bad as it seems. The input consists of n curves, that is n univariate polynomials of degree d and bit-size τ, so the input is $\mathcal{O}(N^3)$. Thus the bit complexity of the algorithm is quadratic with respect to the input.

Acknowledgments. The authors are grateful to Fabrice Rouillier for various discussions and suggestions. We also thank M. Hemmer, E. Berberich, M. Kerber, and S. Limbach for fruitful discussion on the kernel developed at MPII and on the experiments.

References

1. Core. `http://cs.nyu.edu/exact/`.
2. GNU multiple precision arithmetic library. `http://gmplib.org/`.
3. Library for multiple-precision floating-point computations. `http://mpfr.org/`.
4. Multiple precision interval arithmetic library. `http://perso.ens-lyon.fr/nathalie.revol/software.html`.
5. CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.
6. J. Abbott. Quadratic interval refinement for real roots. In *ISSAC 2006, poster presentation*, 2006. http://www.dima.unige.it/ abbott/.
7. E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. EXACUS: Efficient and Exact Algorithms for Curves and Surfaces. In *ESA*, volume 1669 of *LNCS*, pages 155–166. Springer, 2005.
8. E. Berberich, M. Hemmer, M. Karavelas, and M. Teillaud. Revision of the interface specification of algebaic kernel. Technical Report ACS-TR-243301-01, 2007.
9. J.-D. Boissonnat and M. Teillaud, editors. *Effective Computational Geometry for Curves and Surfaces*. Mathematics and Visualization. Springer, 2007.
10. G. Collins, J. Johnson, and W. Krandick. Interval Arithmetic in Cylindrical Algebraic Decomposition. *Journal of Symbolic Computation*, 34(2):145–157, 2002.
11. M. de Berg, M. van Kreveld, M. Overmars, and O. Cheong. Computational Geometry: Algorithms and Applications. 2000.

12. D. I. Diochnos, I. Z. Emiris, and E. P. Tsigaridas. On the complexity of real solving bivariate systems. In C. W. Brown, editor, *Proc. Int. Symp. Symbolic and Algebraic Computation*, pages 127–134, Waterloo, Canada, 2007.
13. A. Edelman and E. Kostlan. How may zeros of a random polynomial are real? *Bulletin of American Mathematical Society*, 32(1):1–37, Jan 1995.
14. A. Eigenwillig, V. Sharma, and C. K. Yap. Almost tight recursion tree bounds for the Descartes method. In *Proc. Int. Symp. on Symbolic and Algebraic Computation*, pages 71–78, New York, NY, USA, 2006. ACM Press.
15. I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards and open curved kernel. In J. Snoeyink and J.-D. Boissonnat, editors, *Proc. 20th Annual ACM Symp. on Computational Geometry (SoCG)*, pages 438–446, New York, USA, Jun 8–11 2004. ACM.
16. I. Z. Emiris, B. Mourrain, and E. P. Tsigaridas. Real Algebraic Numbers: Complexity Analysis and Experimentation. In P. Hertling, C. Hoffmann, W. Luther, and N. Revol, editors, *Reliable Implementations of Real Number Algorithms: Theory and Practice*, LNCS (to appear). Springer Verlag, 2006. also available in www.inria.fr/rrrt/rr-5897.html.
17. E. Fogel, D. Halperin, L. Kettner, M. Teillaud, R. Wein, and N. Wolpert. Arrangements. In J.-D. Boissonnat and M. Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*, Mathematics and Visualization, chapter 1. Springer, 2006.
18. K. Geddes, S. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Pub, 1992.
19. M. Hemmer and S. Limbach. Benchmarks on a generic univariate algebraic kernel. Technical Report ACS-TR-243306-03, 2006.
20. V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core Library For Robust Numeric and Geometric Computation. In *Proc. 15th Annual ACM Symp. on Comp. Geometry*, pages 351–359, 1999.
21. J. Keyser, T. Culver, M. Foskey, S. Krishnan, and D. Manocha. ESOLID - a system for exact boundary evaluation. *Computer-Aided Design*, 36(2):175–193, 2004.
22. J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: a library for efficient and exact manipulation of algebraic points and curves. *Proceedings of the fifteenth annual symposium on Computational geometry*, pages 360–369, 1999.
23. B. Mourrain, P. Pavone, P. Trébuchet, E. P. Tsigaridas, and J. Wintz. SYNAPS, a library for dedicated applications in symbolic numeric computations. In M. Stillman, N. Takayama, and J. Verschelde, editors, *IMA Volumes in Mathematics and its Applications*, pages 81–110. Springer, New York, 2007.
24. F. Rouillier and Z. Zimmermann. Efficient isolation of polynomial's real roots. *J. of Computational and Applied Mathematics*, 162(1):33–50, 2004.
25. RS - A software for real solving of algebraic systems. F. Rouillier. `http://fgbrs.lip6.fr`.
26. R. Wein. High-Level Filtering for Arrangements of Conic Arcs. *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 884–895, 2002.
27. R. Wein and E. Fogel. The new design of CGAL's arrangement package. Technical report, Tel-Aviv University, 2005. `http://www.cs.tau.ac.il/~wein/publications/pdfs/Arr_new_design.pdf`.